Section 1

Fourier transform is a useful tool for processing signals for purposes like feature extraction, data compression, and many other goals.

The only way to process these data and signals is to save them in digital form. So it is necessary to use a digital form of Fourier transform. This form of transformation is called a discrete Fourier transform or DFT and is as follows:

$$x_k = \sum_{n=0}^{N-1} x_n e^{-i2\pi kn/N} \qquad (1)$$

In this section, the DFT of an input signal is computed using a code function that easily uses the above formula. This function can be represented in python as

```
def slow_dft(x):
    "DFT using the raw DFT formula"
    x = np.asarray(x, dtype=float)
    N=len(x)
    n=np.arange(N)
    kn=(n.reshape((N,1)))*n

    E=np.exp(-2j*np.pi*kn/N)
    return np.dot(x,E)
```

Here, the time elapsed to compute the result is compared with python built-in fft. The results are as follows:

TABLE I
Comparison of slow DFT and python built-in DFT

DFT version	Time
slow_dft	107 ms
built-in fft	43 µs

In comparison with the python built-in fft, this code is too slow and has to be improved. It's obvious that using a slow algorithm can cost too much unnecessary computation.

Section 2

It is possible to decrease the computational time by making the matrices smaller. The DFT formula can be divided into two terms by computing DFT of even and odd part of the signal. This algorithm can go on until the length of sub-signals (even and odd parts) is even. So it is possible to use a recursive function and divide the signal into two parts till it's possible. The code is as follows:

```
def FFT(x):
    "DFT by dividing the raw DFT formula"
    x = np.asarray(x, dtype=float)
    N=len(x)

    if N%2==0:
        x_even=x[::2]
        x_odd=x[1::2]
        F1=FFT(x_even)
        F2=FFT(x_odd)
        k=np.arange(N)
        p=np.exp(-2j*np.pi*k/N)
        return np.concatenate([F1+p[:int(N/
2)]*F2, F1+p[int(N/2):]*F2])

    else:
        return slow_dft(x)
```

The speed of the algorithm is improved with this simple change in the main DFT formula. The results are shown below:

TABLE II
Comparison of raw FFT and python built-in DFT

DFT version	Time
slow_dft	107 ms
FFT	43 ms
built-in fft	43 µs

The algorithm has been faster and uses fewer computations. Still, the algorithm can be improved.

Section 3

By dividing the input signal into 32 vectors, instead of using recursive functions, the algorithm can be even faster. So the main formula can be re-expressed as follows:

$$N = h * 32$$

$$\begin{aligned} x_k &= \sum_{n=0}^{N-1} x_n e^{-\frac{i2\pi kn}{N}} \\ &= \sum_{n=0}^{31} x_n e^{-\frac{i2\pi kn}{N}} + \sum_{n=32}^{63} x_n e^{-\frac{i2\pi kn}{N}} + \cdots \\ &= \sum_{n=0}^{31} x_n e^{-\frac{i2\pi kn}{N}} + \sum_{n=0}^{31} x_{n+32} e^{-\frac{i2\pi k(n+32)}{N}} + \cdots \\ &= \sum_{n=0}^{31} x_n e^{-\frac{i2\pi kn}{N}} + e^{-\frac{i2\pi k\times 32}{N}} \sum_{n=0}^{31} x_{n+32} e^{-\frac{i2\pi kn}{N}} + \cdots \end{aligned} \tag{2}$$

According to the above expansion of DFT, there is a similarity between each term and it can be represented as a multiplication of matrices. So it is possible to use some exponential parts several times and the vectors can be converted to matrices. It means that the computations will be less. The algorithm based on the above expansion is

```
def FFT_vect(x):
    "DFT by vectorizing the DFT formula"
    x = np.asarray(x, dtype=float)
    N=len(x)

    if np.log2(N)%1==0:
        N_min=min(N,32)
        x=x.reshape((-1,N_min))
        n=np.arange(N_min)
        k=np.arange(N)[:,np.newaxis]
        M=np.exp(-2j*np.pi*n*k/N)
        F=np.matmul(M,np.transpose(x))
        d=np.arange(int(N/N_min))*N_min
        factor=np.exp(-2j*np.pi*k*d/N)
        X=F*factor
        output=sum(np.transpose(X))
        return output
else:
        return slow_dft(x)
```

As it can be seen in the code, the expansion could be represented as some multiplications of matrices.

Here the results show that the algorithm had a remarkable improvement.

TABLE III
Comparison of vectorized FFT and python built-in DFT

DFT version	Time
slow_dft	107 ms
FFT	43 ms
FFT_vectorized	6.2 ms
built-in fft	42 µs

For sure, the results of the functions are compared with the python built-in fft and the outputs are the same.